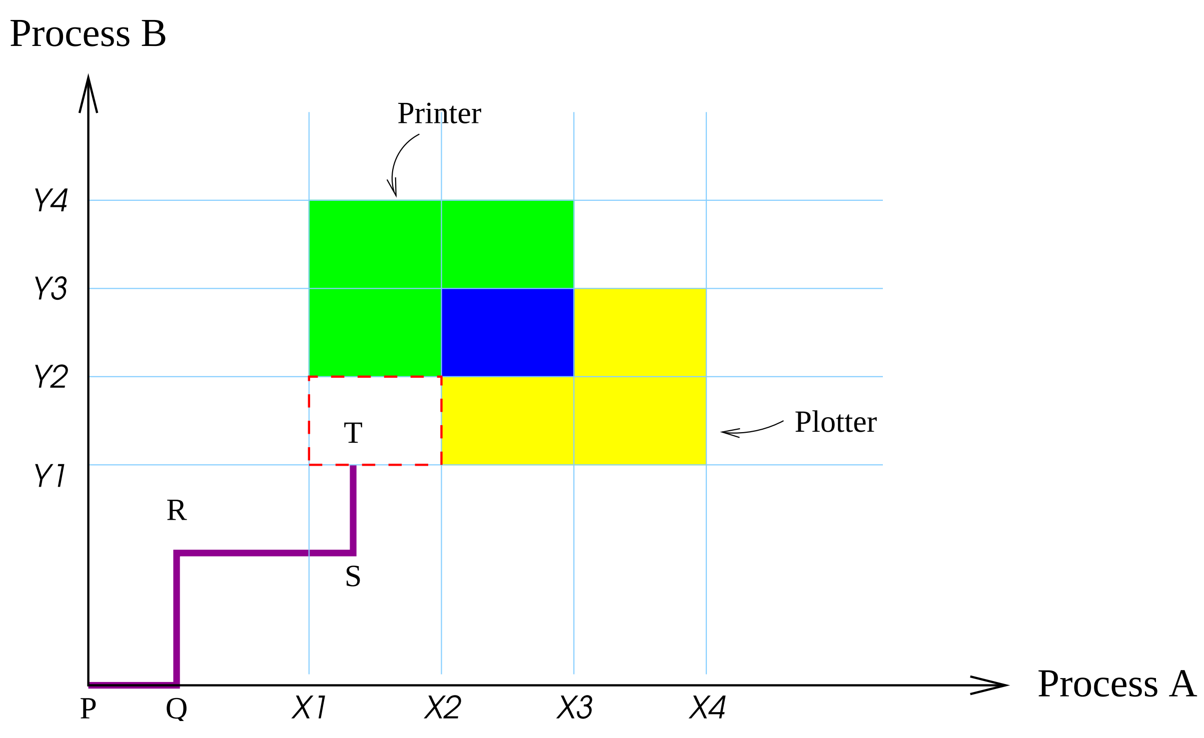
Deadlock Avoidance

- If we can detect deadlocks, can we avoid them?
- Yes, but. . .
- We can avoid deadlocks if certain information is available in advance

1 / 38

Process Trajectories



A needs the printer from X_1 to X_3; it needs the plotter from X_2 to X_4.
B needs the plotter from Y_1 to Y_3; it needs the printer from Y_2 to Y_4.

2 / 38

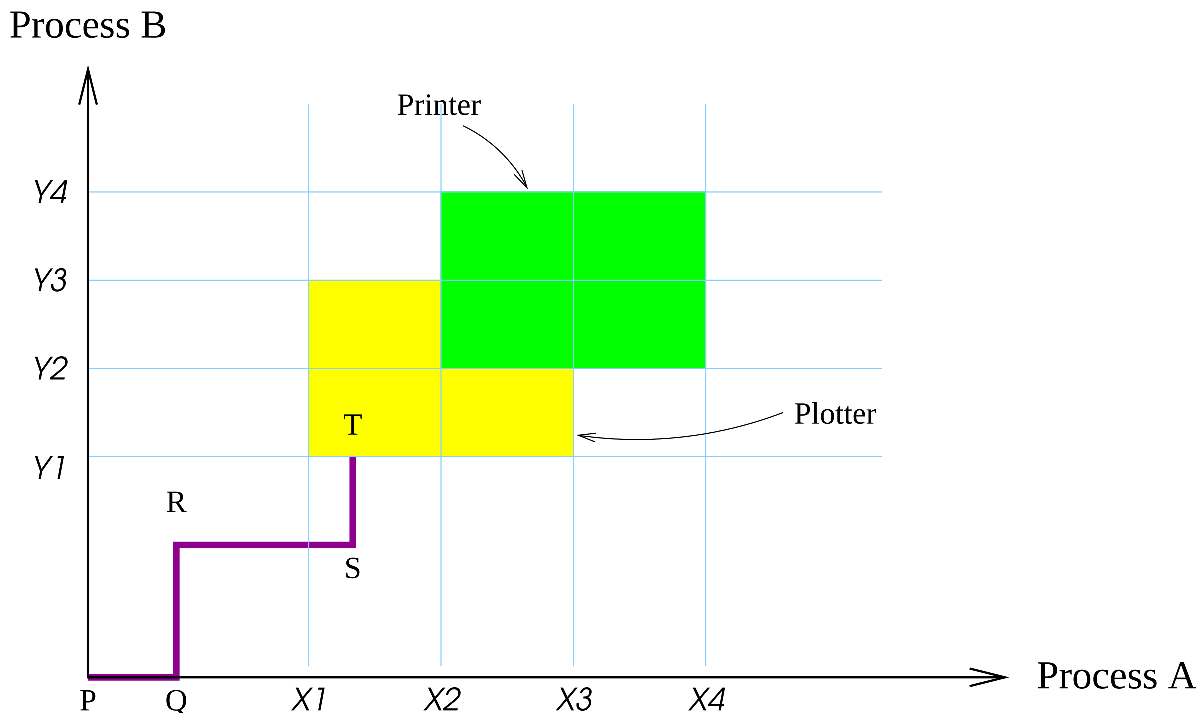
Problems

- Warning sign: A and B are asking for resources in a different order
- Green region: both A and B have the printer — impossible
- Yellow region: both have the plotter
- Blue — both have both devices. . .
- The colored regions represent impossible states and cannot be entered

Avoiding Deadlock

- If the system ever enters the red-bordered state, it *will* deadlock
- At time t, cannot schedule B
- If we do, system will enter deadlock state
- Must run A until X_4

Alternate Process Trajectory



If A and B ask for the plotter first, there's no danger. B will clearly block if it's scheduled, so A will proceed. The dangerous state was where a process entered a *clear* box that would deadlock.

5 / 38

Safe and Unsafe States

- A state is *safe* if not deadlocked and there is a scheduling order in which all processes can complete, even if they all ask for all of their resources at once
- An *unsafe* state is not deadlocked, but no such scheduling order exists
- It may even work, if a process releases resources at the right time

6 / 38

The Banker's Algorithm (Dijkstra, 1965)

- Assume we're dealing with a single resource — perhaps dollars
- Every customer has a "line of credit" — a *maximum possible resource allocation"*
- The banker only has a certain amount of cash on hand
- Not everyone will need all of their credit at once
- Solution: only grant requests if they leave us in a safe state

Example

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

The first state is safe; we can grant the requests sequentially. The second state is safe; we can grant C's maximum request, let it run to completion, and then have $4 to give to B or D.

If, after the second state, we give $1 to B, we enter an unsafe state — there isn't enough money left to satisfy all possible requests.

The Banker's Algorithm for Multiple Resources

- Build matrices C (currently assigned) and R (and still needed), just as we used for deadlock detection
- Build vectors E (existing resources) and A (available), again as before
- To see if a state is safe:
  1. Find a row R whose unmet needs are $\leq A$
  2. Mark that row; add its resources to A
  3. Repeat until either all rows are marked, in which case the state is safe, or some are unmarked, in which case it's unsafe
- Run this algorithm any time a resource request is made

9 / 38

Why Isn't This Useful?

- Every process must state its resource requirements at startup
- This is rarely possible today
- Processes come and go
- Resources vanish as hardware breaks
- Not really used these days...

10 / 38

Example: File Binding Time

- Old mainframes: *all* file name binding is done immediately prior to execution. Also makes it easy to move files around
- Classic Unix: file names on command line (but not clearly identifiable as such) or compiled-in to commands. Occasional overrides via environment variables or options.
- GUIs: many files selected via menus

Early versus late binding is a major issuse in system design. Both choices here have their advantages and disadvantages

Deadlock Prevention

Preventing Deadlocks

- Practically speaking, we can't avoid deadlocks
- Can we prevent them *in the real world*?
- Let's go back to the four conditions:
  1. Mutual exclusion
  2. Hold and Wait
  3. No preemption
  4. Circular wait

Attacking Mutual Exclusion

- Much less of an issue today — fewer single-user resources
- Many of the existing ones are dedicated to single machines used by single individuals, i.e., CD drives
- Printers are generally spooled
  - ⇒ No contention; only the printer daemon requests it
- Not a general solution, but useful nevertheless

13 / 38

Attacking Holding and Wait

- Could require processes to state their requirements up front
- Still done sometimes in the mainframe world
- Of course, if we could do that, we could use the Banker's Algorithm

14 / 38

A Variant is Useful

- Before requesting a resource, release *all* currently-held resources
- Request all new ones at once
- Doesn't work if some resources *must* be held

15 / 38

Attacking Circular Wait

- Number each possible resource:
  1. Scanner
  2. Printer
  3. Tape drive
  4. ...
- Resources must be requested in numerical order
- Can't deadlock — prevents the out-of-order scenario we saw earlier
- Used on old mainframes
- Can combine this with release-and-rerequest

16 / 38

Mainframe Resources

- Wait until enough tape drives are available
- Wait until memory region is available
- Wait for all disk files to be free

Order based on typical wait time — disk files freed up quickly, while tape drives waited for operators to find and mount tape reels

17 / 38

Two-Phase Locking

- Frequently used in databases
- Processes need to lock several records then update them all
- Phase 1: try locking each record, one at a time
- On failure, release them all and restart
- When they're all locked, do the updates and then the release
- Effectively the same as "request everything up front"

18 / 38

What's Really Done About Deadlocks?

- In the OS, nothing...
- Overprovision some resources, such as process slots
- But — still very important in some applications, notably databases

19 / 38

What About Linux?

- No deadlock prevention or detection for applications or threads
- The kernel does care about deadlocks for itself.

```
/* We can't just spew out the rules
 * here because we might fill the
 * available socket buffer space and
 * deadlock waiting for auditctl to
 * read from it... which isn't ever
 * going to happen if we're actually
 * running in the context of auditctl
 * trying to _send_ the stuff */
```

20 / 38

Scheduling

- Suppose several processes are runnable?
- Which one is run next?
- Many different ways to make this decision

21 / 38

Environments

- Old batch systems didn't have a scheduler; they just read whatever was next on the input tape
- Actually, they did have a scheduler: the person who loaded the card decks onto the tape
- Hybrid batch/time-sharing systems tend to give priority to short timesharing requests
- Still a policy today: must give priority to interactive requests

22 / 38

Process Behavior

- Processes alternate CPU use with I/O requests
- I/O requests frequently block, either waiting for input or when too much has been written and no buffer space is available
- CPU-bound processes think more than they read or write
- I/O-bound processes do lots of I/O; it's (usually) not that the I/O operations are so time-consuming
- Absolute speed of CPU and I/O devices is irrelevant; what matters is the *ratio*
- CPUs have been getting much faster relative to disks

When to Make Scheduling Decisions

- After a fork — run the parent or child?
- On process exit
- When a process blocks
- When I/O completes
- Sometimes, after timer interrupts

Preemptive vs. Nonpreemptive Schedulers

- Nonpreemptive scheduler: lets a process run as long as it wants
- Only switches when it blocks
- Preemptive: switches after a time quantum

25 / 38

Categories of Scheduling Algorithms

- Batch — responsiveness isn't important; preemption moderately important
- Interactive — must satisfy a human; preemption important
- Real-time — often nonpreemptive

26 / 38

Goals

- Fairness — give each process its share of the CPU
- Policy and enforcement — give preference to work that is administratively favored; prevent subversion of OS scheduling policy
- Balance — keep all parts of the system busy

27 / 38

Goals: Batch Systems

- Throughput — maximize jobs/hour
- Turnaround time — return jobs quickly. Often want to finish short jobs very quickly
- CPU utilization

28 / 38

Interactive Systems

- Response time — respond quickly to user requests
- Meet user expectations — psychological
  - Users have a sense of "cheap" and "expensive" requests
  - Users are happier if "cheap" requests finish quickly
  - "Cheap" and "expensive" don't always correspond to reality!

29 / 38

Real-Time Systems

- Meet deadlines — avoid losing data (or worse!)
- Predictability — users must *know* when their requests will finish
- Requires careful engineering to match priorities to actual completion times and available resources

30 / 38

Batch Schedulers

- First-come, first-served
- Shortest first
- Shortest remaining time first
- Three-level scheduler

31 / 38

First-Come, First-Served

- Run the first process on the run queue
- Never preempt based on timer
- Seems simple; just like waiting in line
- Not very fair

32 / 38

First-Come, First-Served

- Imagine a CPU-bound process A: thinks for 1 second, then reads 1 disk block
- There's also an I/O-bound process B that needs to read 1000 blocks
- A runs for 1 second, then issues an I/O request
- B runs for almost no time, then issues an I/O request
- A then runs for another second
- It takes 1000 seconds for B to finish

33 / 38

Shortest First

- Suppose you know the time requirements of each job
- A needs 8 seconds, B needs 4, C needs 4, D needs 4
- Run B, C, D, A
- Nonpreemptive
- Provably fair:
  - Suppose four jobs have runtimes of a, b, c, and d
  - First finishes at time a, second at $a + b$, etc
  - Mean turnaround is $(4a + 3b + 2c + d)/4$
  - d contributes less to the mean

34 / 38

Optimality Requires Simultaneous Availability

- Jobs don't all arrive at the same time
- Can't make optimal scheduling decision without complete knowledge
- Example: jobs with times of 2,4,1,1,1 that arrive at times 0,0,3,3,3
- Shortest-first runs A, B, C, D, E; average wait is 4.6 secs
- If we run B, C, D, E, A, average wait is 4.4 secs
- While B is running, more jobs arrive, allowing a better decision for the total load

Shortest Remaining Time Next

- Preemptive variant of FCFS
- Still need to know run-times in advance
- Helps short jobs get good service
- May have a problem with indefinite overtaking

Three-Level Scheduler

- First stage: job queue
- Select different types of jobs (i.e., I/O- or CPU-bound) to balance workload
- Note: relies on humans classify jobs in advance
- Second stage: availability of main memory

⇒ Closely linked to virtual memory system; let's defer that

- CPU scheduler

User Requirements

- Users must be able to specify job characteristics: estimated CPU time, I/O versus CPU balance, perhaps memory
- Scheduler categories must reflect technical and managerial issues
- Lying about characteristics may give better turnaround times, but at hte expense of total system throughput
- Should the resonse be technical or administrative?